

APPLICATION FOR UNITED STATES LETTERS PATENT

For

**HARDWARE-BASED MULTI-THREADING FOR PACKET
PROCESSING**

Inventors:

Yatin Hoskote
Sriram R. Vangal
Vasanth K. Erraguntla
Nitin Y. Borkar

Prepared by:

BLAKELY SOKOLOFF TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard
Los Angeles, CA 90025-1026
(206) 292-8600

Attorney's Docket No.: 42P18633

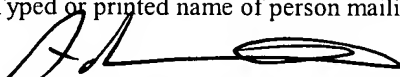
"Express Mail" mailing label number: EV320119501US

Date of Deposit: March 31, 2004

I hereby certify that I am causing this paper or fee to be deposited with the United States Postal Service "Express Mail Post Office to Addressee" service on the date indicated above and that this paper or fee has been addressed to Mail Stop Patent Application, Commissioner for Patents, PO Box 1450, Alexandria, VA 22313-1450.

Adrian Villarreal

(Typed or printed name of person mailing paper or fee)



(Signature of person mailing paper or fee)

March 31, 2004

(DATE SIGNED)

HARDWARE-BASED MULTI-THREADING FOR PACKET PROCESSING

FIELD OF THE INVENTION

- 5 **[0001]** The field of invention relates generally to TCP packet processing and, more specifically but not exclusively relates to techniques for processing TCP packets using hardware-based multi-threading.

BACKGROUND INFORMATION

- 10 **[0002]** Transmission Control Protocol (TCP) is a connection-oriented reliable protocol accounting for over 80% of today's network traffic. TCP exists within the Transport layer, between the Application layer and the Internet Protocol (IP) layer, providing a reliable and guaranteed delivery mechanism to a destination machine. Connection-oriented protocols guarantee the delivery of packets by tracking the
15 transmission and receipt of individual packets during communication. A session is able to track the progress of individual packets by monitoring when a packet is sent, in what order it was sent, and by notifying the sender when it is received so it can send more. To support this functionality, a significant level of processing must be performed at both the sending and destination machines.
- 20 **[0003]** Today, TCP processing is performed almost exclusively through software. Even with the advent of GHz (gigahertz) processor speeds, there is a need for dedicated processing in order to support high bandwidths of 10 gigabits per second (Gbps) and beyond. Several studies have shown that even state-of-the-art servers are forced to completely dedicate their CPUs (central processing unit) to TCP
25 processing when bandwidths exceed 1Gbps. At 10Gbps, there are 14.8 million minimum-size Ethernet packets arriving every second, with a new packet arriving every 67.2 nanoseconds (ns). Allowing a few nanoseconds for overhead, wire-speed TCP processing requires several hundred instructions to be executed

approximately every 50ns. Given that a majority of TCP traffic is composed of small packets, this is an overwhelming burden on a server's CPU.

[0004] A generally accepted rule of thumb for network processing is that 1GHz CPU processing frequency is required for a 1Gbps Ethernet link. For smaller packet sizes on saturated links, this requirement is often much higher. Ethernet bandwidth is slated to increase at a much faster rate than the processing power of leading edge microprocessors. Clearly, general-purpose processors will not be able to provide the required computing power in coming generations.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] The foregoing aspects and many of the attendant advantages of this invention will become more readily appreciated as the same becomes better understood by reference to the following detailed description, when taken in conjunction with the accompanying drawings, wherein like reference numerals refer to like parts throughout the various views unless otherwise specified:

[0006] Figure 1a is a schematic diagram illustrating data buffering and copying associated with conventional processing of an outbound TCP packet;

[0007] Figure 1b is a schematic diagram illustrating data buffering and copying associated with conventional processing of an inbound TCP packet

[0008] Figure 2 is a schematic diagram illustrating an architecture of a multi-thread TCP offload engine (TOE) and associated peripheral circuitry used for performing hardware-based TCP input and output processing, according to one embodiment of the invention;

[0009] Figure 3 is a schematic diagram showing details of the processing engine of Figure 2, according to one embodiment of the invention;

[0010] Figure 4a is a packet processing pipeline diagram illustrating operations performed during input processing of inbound TCP packets, according to one embodiment of the invention;

[0011] Figure 4b is a packet processing pipeline diagram illustrating operations performed during output processing of outbound TCP packets, according to one embodiment of the invention;

[0012] Figure 5 is a flowchart illustrating operations and logic performed in accordance with the packet processing pipeline diagram of Figure 4a;

[0013] Figure 6 is a flowchart illustrating operations and logic performed in accordance with the packet processing pipeline diagram of Figure 4b;

[0014] Figure 7 illustrates a set of general purpose instruction and special purpose instructions that may be implemented by an exemplary multi-threaded TOE, according to one embodiment of the invention;

[0015] Figure 8 is a block schematic diagram of the scheduler of Figure 2,
5 according to one embodiment of the invention; and

[0016] Figure 9 is a graph comparing predicted bandwidth performance of a multi-threaded TOE and a single-threaded TOE, according to an exemplary Ethernet full duplex performance model.

DETAILED DESCRIPTION

[0017] Embodiments of methods, apparatus and systems for processing TCP
5 packets via multi-threaded hardware techniques are described herein. In the
following description, numerous specific details are set forth to provide a thorough
understanding of embodiments of the invention. One skilled in the relevant art will
recognize, however, that the invention can be practiced without one or more of the
specific details, or with other methods, components, materials, *etc.* In other
10 instances, well-known structures, materials, or operations are not shown or
described in detail to avoid obscuring aspects of the invention.

[0018] Reference throughout this specification to “one embodiment” or “an
embodiment” means that a particular feature, structure, or characteristic described in
connection with the embodiment is included in at least one embodiment of the
15 present invention. Thus, the appearances of the phrases “in one embodiment” or “in
an embodiment” in various places throughout this specification are not necessarily
all referring to the same embodiment. Furthermore, the particular features,
structures, or characteristics may be combined in any suitable manner in one or
more embodiments.

[0019] In accordance with aspects of the embodiments described herein, a novel
20 packet processing architecture and associated techniques for TCP termination (i.e.,
input and output processing) for multi-gigabit Ethernet traffic are disclosed. The
architecture includes a dedicated TCP offload engine (TOE) that implements a
multiple thread mechanism entirely in hardware, including thread suspension,
25 scheduling, and save/restore of thread state.

[0020] In order to better understand and appreciate advantages of the TOE
architecture, a brief discussion of the limitations and problems associated with the
conventional software-based TCP termination scheme is first discussed.

[0021] There are several operations in TCP termination that require improvement if future increases in bandwidth are to be handled efficiently. A first consideration concerns minimizing intermediate copies of data during both transmits and receives, which represents a significant performance bottleneck under the conventional software architecture. For examples, Figures 1a and 1b respectively show software transmit and receive paths corresponding to conventional packet processing. During each set of packet processing, multiple copies of the same data are copied between buffers, creating redundant operations.

[0022] Each of Figures 1a and 1b show a conventional system architecture including one or more processors 100, a memory controller hub (MCH) 102, host memory 104, and a network interface controller (NIC) 106. The processors 100 are communicatively coupled to MCH 102 via a front-side bus (FSB) 108. The host memory 104 is communicatively coupled to MCH 102 via a memory bus 110. The NIC 106 is communicatively coupled to MCH 102 via a peripheral bus, such as a peripheral component interconnect (PCI) bus 112 or a PCI Express (PCI-X) bus.

[0023] Under the conventional software transmit path of Figure 1a, the process is initiated by an application hosted by an operating system (OS) running on a processor 100. The OS maintains an application buffer 114 for the application in host memory 104; this buffer is typically associated with an OS user space, which occupies a separate portion of memory from an OS kernel. The data to be transmitted out via NIC 106 is initially stored in application buffer 114.

[0024] Under a first operation (depicted by an encircled "1"), a processor read operation is performed, wherein the data to be transferred are read from application buffer 114. The data are then copied (i.e., written) to a socket buffer 116 in host memory 104 during a second operation. The socket buffer 116 corresponds to a network protocol component (e.g., a TCP/IP driver) running in the OS kernel. The network protocol component assembles the data into a plurality of packets based on

the protocol to be employed for the network transfer (e.g., TCP). During a third operation, the packets 118 are then transferred to a transmit (Tx) buffer 120 hosted by NIC 106 via a DMA (direct memory access) write transfer. Subsequent, NIC 106 transfers packets 118 from transmit buffer 120 to a first hop in a network via which
5 the destination node for the transmission may be reached.

[0025] As depicted in Figure 1b, the conventional software receive processing operations are roughly analogous to the transmit operations, only in reverse. In this instance, the plurality of packets 118 are received at NIC 106 and stored in a receive (Rx) buffer 122. However, in this instance, the order of the packets may be mis-
10 ordered relative to the original order of the packets when they were sent from the sending machine. Typically, the mis-ordered packets will be transferred from Rx buffer 122 to socket buffer 116 via a DMA read transfer, whereupon they will be reordered and reassembled by the OS kernel network protocol component. The assembled data are then transferred to application buffer 114 via sequential
15 processor read and processor write operations.

[0026] The foregoing conventional scheme is highly inefficient. A more efficient mechanism for transferring data between application buffers and the NIC would be advantageous, both to improve performance and reduce traffic on the front-side bus. In one embodiment, this is achieved by pre-assigning buffers for data that is
20 expected to be received, as described below in further detail.

[0027] Another problem concerns memory accesses and associated latency. Processing transmits and receives requires accessing context information for each connection that may be stored in host memory. Each memory access is an expensive operation, which can take up to 100ns. The TOE architecture optimizes
25 the TCP stack to reduce the number of memory accesses, significantly increasing performance. At the same time, the TOE architecture employs techniques to hide memory latency via multi-threading.

[0028] It also would be advantageous to provide improved access to state information. The context information for each TCP connection is of the order of several hundred bytes. Some method of caching the context for active connections is necessary. Studies have shown that caching context for a small number of connections is sufficient (burst mode operation) to see performance improvement (See, K. Kant, "TCP offload performance for front-end servers", to appear in *proc. of GLOBECOM 2003*, Dec 2003, San Francisco, CA.). Increasing the cache size beyond that does not help unless it is made large enough to hold the entire allowable number of connections. Protocol processing requires frequent and repeated access to various fields of each context. A mechanism, such as fast local registers, to access these fields quickly and efficiently reduces the time spent in protocol processing. In addition to context information, these registers can also be used to store intermediate results during processing.

[0029] Current instruction execution schemes are inefficient. Reducing the number of instructions to be executed by optimizing the TCP stack would go a long way in reducing the processing time per packet. Another source of overhead that reduces host efficiency is the communication interface between the host and NIC. For instance, an interrupt driven mechanism, such as that conventionally used by PCI-based architectures, tends to overload the host and adversely impact other applications running on the host. Other network transmission-related processing, such as encryption, decryption, classification, etc., may be more-efficiently performed via hardware than conventional software-based schemes.

Architecture Details

[0030] Figure 2 shows a top-level architecture diagram 200 of a computer platform 202 including an implementation of a multi-threaded TOE, according to one embodiment. In one embodiment, as depicted, the TOE functionality is implemented via a platform chipset integrated circuit (IC) 204. As used herein the term "chipset"

may include one or more physical integrated circuit chips. The exemplary computer platform also includes one or more processors 206 connected to a front-side bus 208, as well as host memory 210 connected to a memory bus 211.

[0031] In general, platform 202 will include at least one of either a network interface card or integrated network interface controller (both referred to as NICs). An exemplary NIC 242 is shown at the lower portion of platform 202. NIC 242 includes an input buffer 244 and an output buffer 246. Typically, NIC 242 will be connected to IC 204 via an expansion bus 250, such as a PCI (peripheral component interconnect) or PCI-X (PCI-Express) bus.

[0032] The TOE architectural components include an execution core (i.e., processing engine) 212, a scheduler 214, a large on-die cache (L3) (depicted as transmission control block (TCB) cache 216), and an integrated DMA controller (depicted as a transmit DMA block 218 and a receive DMA block 220). Optionally, the DMA controller may comprise separate circuitry on the same or different IC chip. In addition, the architecture provides well-defined interfaces to NIC 242, host memory 210 and the one or more host processors 208 via a host interface 222, a host memory interface 223, and a NIC interface 248, respectively.

[0033] In one embodiment, the architecture presents three queues as a hardware mechanism to interface with the one or more host processors 206 via host interface 222. A doorbell queue (DBQ) 224 is used to initiate send (or receive) requests. A completion queue (CQ) 226 and an exception/event queue (EQ) 228 are used to communicate processed results and events back to the host processor(s). In one embodiment, the architecture also defines additional queues including a transmit queue 230, a header and data queue 232, and a memory queue 234.

[0034] A timer unit 236 provides hardware offload for frequently used timers associated with TCP processing. In one embodiment, the timer unit 236 supports

four timers, while in another embodiment seven timers are supported. In one embodiment, the TOE also includes hardware assist for virtual-to-physical (V2P) address translation, as depicted by a V2P block 238. In one embodiment, the TOE may also include provisions for supporting IP security (IPSec) functions, as depicted
5 by an IPSec block 240.

[0035] Processing engine 212 comprises a high-speed processing engine, which includes interfaces to the various peripheral units. In one embodiment, a dual-frequency design is used, with the processing engine clocked several times faster (core clock) than the peripheral units (slow clock). In one embodiment, the clock
10 speed for processing engine 212 is 4.8 GHz, while the clock speed for peripheral units including scheduler 214 and TCB cache 216 is 1.2 GHz. This approach results in minimal input buffering needs, enabling wire-speed processing.

[0036] In one embodiment, TCB cache 216 comprises 1MB of on-die cache to store TCP connection context data, which provides temporal locality for 2048
15 connections, with additional context data residing in host memory 210. The context data comprises a portion of the transmission control block (TCB) that TCP is required to maintain for each connection. Caching this context on-chip is critical for 10Gbps performance. In addition, to avoid intermediate packet copies on receives and transmits, the integrated direct memory access (DMA) controller (i.e., transmit
20 DMA block 218 and receive DMA block 220) enables a low-latency transfer path and supports direct placement of data in application buffers without substantial intermediate buffering. Scheduler 214 provides global control to the processing engine 212 at a packet level granularity. Scheduler 214 also operates as a "traffic cop," directing various data to appropriate destinations.

[0037] In one embodiment, the DMA controller supports four independent,
25 concurrent channels and provides a low-latency/high throughput path to/from various memory stores and buffers. In one embodiment, the TOE constructs a list of

descriptors (commands for read and write), programs the DMA controller, and initiates the DMA start operation. In response, the DMA controller transfers data from sources to destinations based on the commands defined for respective descriptors in the list. Upon completion of the commands, the DMA controller
5 notifies the TOE, which updates completion queue 226 to notify host processor 208 of the result.

[0038] A micro-architecture block diagram of one embodiment of processing engine 212 is detailed in Figure 3. The micro-architecture features a high-speed fully pipelined ALU 300 at its heart, communicatively coupled to a wide working
10 register 302 and a core receive queue 304 via multiplexers 306A and 306B and buses 308A and 308B. In one embodiment, the wide working register is 512 bytes (B) wide. In one embodiment, buses 308A and 308B are 32-bits wide. TCB context data for the current scheduled active connection is loaded into wide working register 302 for processing. The execution core (ALU 300) performs TCP
15 processing under direction of instructions issued by an instruction cache 310. A control instruction is read every core cycle and loaded into an instruction register (IR) 312. The execution core reads instructions from IR 312, decodes them if necessary, and executes them every cycle. The functional units in the core include arithmetic and logic units, shifters and comparators - all optimized for high frequency
20 operation. The core includes a large register set, including two 256B register arrays (i.e., scratch registers 314) to store intermediate processing results. The scheduler 214 exercises additional control over execution flow via various control inputs entered via instruction cache 310.

[0039] In an effort to hide host and TCB memory latency and improve throughput,
25 in one embodiment the engine is multi-threaded. To support multi-threading, the design includes a thread cache 316, running at core speed, which allows intermediate architecture state to be saved and restored for each thread. In one

embodiment, thread cache 316 is estimated to be 8-16 threads deep and 512 bytes wide. The width of the cache is determined by the amount of context information that needs to be saved for each packet. The depth of the cache is determined by the packet arrival rate. Analysis shows that for 256 byte packets on a 10Gbps link
5 for performing both receives and transmits, a 16 deep cache is sufficient because that is more than the number of packets that could be active (i.e., being processed) at any point in time.

[0040] The micro-architecture design also provides a high-bandwidth connection 318 between the thread cache and working register 302, making
10 possible very fast and parallel transfer of thread state between the working register and thread cache 316. Thread context switches can occur during both receives and transmits and when waiting on outstanding memory requests or on pending DMA transactions. This ensures that the overhead penalty from thread switches is minimal. At the exemplary sample frequencies shown in the Figures herein, the
15 overhead penalty is less than 3ns. The working register 302, execution core and scratch registers 314 are completely dedicated to the packet currently being processed. This is different from other conventional approaches where the resources are split up *a priori* and dedicated to specific threads via software control. This ensures adequate resources for each packet without having to duplicate
20 resources and increase engine die area.

[0041] In one embodiment, processing engine 212 features a cacheable control store, which enables only code relevant to specific TCP processing operations to be cached, with the rest of the code in host memory 210. A good replacement policy allows TCP code in the instruction cache 310 to be swapped as required. This also
25 provides flexibility and allows for easy protocol updates. In one embodiment, TCP code is loaded into host memory 210 during system initialization operations. For

example, in one embodiment the TCP code is included as part of an operating system TOE driver.

Input Processing

[0042] Figure 4a shows a packet processing pipeline diagram corresponding to packet receive operations, according to one embodiment. A corresponding flowchart illustrating operations and logic performed during input processing is shown in Figure 5. The first set of input processing operations pertain to NIC processing. In a block 500, inbound packets from NIC interface 242 are buffered in header and data queue 232, which in one embodiment functions as a queued inbound buffer. A splitter parses the packet contents to separate the header from the payload (i.e., data), forwarding the header to scheduler 214. In addition, conventional NIC processing operations are performed, including processing a NIC descriptor (containing data pertaining to the TCP connection), performing packet validation checks, and performing a TCP checksum check. If the packet is determined to be invalid via these checks, the packet is discarded.

[0043] The next set of operations pertain to TCB cache 216 and/or host memory 210 accesses. In general, these operations are used to correlate a given packet with its corresponding TCP connection. First, a check to see if TCP connection context data corresponding to the TCP connection used to deliver the packet is available in the TCB cache or host memory. If not, a new context entry is made in memory and copied to the TCB cache.

[0044] In one embodiment, TCB cache 216 is configured as a segmented cache, wherein each segment is accessed via a corresponding hash result. Generally, the number of segments in the TCB cache will depend on the granularity of the hash algorithm employed. Accordingly, in a block 501 connection identification (ID) data is extracted from the NIC descriptor. In one embodiment, the connection ID data comprise the addresses of the sender and destination corresponding to the TCP

connection used to deliver the packet. A hash is then performed on the connection ID data using a pre-defined hashing algorithm.

5 **[0045]** The result of the hash will produce a hashed segment index into TCB cache 216. Thus, using this result, a hash-based lookup is performed against TCB cache 216. If the hashed segment index exists in the TCB cache (indicating corresponding TCP connection context data are present in the TCB cache), a cache hit will result. If not, a cache miss results. The result of a cache hit or miss at this point is depicted by a decision block 502 in Figure 5.

10 **[0046]** On a cache hit, the TCP connection context data (corresponding to the currently-processed packet) is loaded into working register 302, as depicted by a block 508. On a miss, a lookup against hashed entries in host memory 210 is scheduled via memory queue 234, and then is performed (generally asynchronously), as depicted by a block 503. As illustrated by a decision block 504, if a hit results (indicating the TCP connection context data are present in host
15 memory), the corresponding data are copied to TCB cache 216 (both TCP connection context data and hash index) in a block 506, and then the context data are loaded into working register 302 in block 508. If a miss results, the TCP connection context data have not been generated for the connection. Accordingly, a context entry (hash index and context data) is created in host memory 210, as
20 depicted by a block 505, and immediately copied to TCB cache 216 in block 506. The logic then proceeds to block 508 to load the context into the working register.

[0047] In a block 510 TCP processing corresponding to the packet is initiated via the TOE. TCP processing includes operations generally associated with performing TCP processing-related activities, including reading doorbell queue 224, creating a
25 timestamp for the packet (for acknowledgement/statistics purposes), parsing the TCP header, updating TCP statistics, generating or updating packet sequence

numbers, scheduling ACK (acknowledgement) messages, etc. These operations are depicted by a block 511.

5 [0048] Substantially in parallel, the execution core also programs the DMA controller (i.e., DMA receive block 220) and queues DMA receive requests in a block 512. Payload data is then transferred from header and data queue 232 to pre-posted locations in host memory 210 using DMA transfer operations in a block 514. This concurrent, low-latency DMA transfer yields enhanced performance, while hiding memory latencies associated with conventional TCP processing techniques.

10 [0049] In the meantime, TCP input processing continues until it is completed, as depicted by a block 516. Subsequent to TCP processing completion, the logic proceeds to a block 518, wherein the TCP connection context is updated with the processing results and written back to the TCB cache 216. In a block 520, the scheduler 214 then updates completion queue 226 with completion descriptors and exception/event queue 228 with completion status data (e.g., pass/fail), which can
15 generate a host processor interrupt, thus informing the processor that processing for a packet has been completed. In general, events can either be exception status events or interrupts. In one embodiment, an operating system-level or firmware driver may be employed to coalesce the events and interrupts for more efficient processing.

20 Output Processing

[0050] Figure 4b shows a packet processing pipeline diagram corresponding to packet transmit operations, according to one embodiment, while corresponding operations and logic are illustrated in the flowchart of Figure 6. During corollary operations (i.e., operations that are not performed by the TOE), the OS employs a
25 TOE driver (implemented as a software-based OS driver or a firmware-based driver) to place doorbell descriptors in doorbell queue 224, as shown in a block 600. The doorbell queue contains pointers to either the Tx or Rx descriptors queues

(depending on whether the doorbell queue entry corresponds to a send or receive operations), which reside in host memory 214. The TOE is responsible for fetching and caching the descriptors in TCB cache 216.

[0051] Next, the connection (corresponding to a given transmit session) is identified by scheduling a lookup against TCB cache 216 in a block 602. In a manner similar to that discussed above, first a lookup is made against the TCB cache. If a hit results, the TCP connection context data is loaded into working register 302 to initiate core TCP processing for the current outbound packet. If a cache miss results, a memory lookup is queued and performed in a manner analogous with block 503. If a memory hit results, the context entry is copied to the TCB cache and loaded into the working register. If a memory miss results, a TCP context entry is created in host memory, copied to the TCB cache, and then loaded into the working register.

[0052] In accordance with a block 604, processing engine 212 then performs the heart of TCP output processing under programmed control at high speed. In one embodiment, these operations include determining route validity, generating TCP sequence numbers, generating TCP headers, setting up runtime timers (by interacting with the timer block to set and clear timers 236), and computing checksums. In parallel with the TCP output processing operations, the core also programs the DMA control unit (i.e., DMA transmit block 218) by building an appropriate descriptor ring and queues the transmit DMA requests in transmit queue 230, as depicted by a block 606. Here again, for low latency, payload data is transferred from the payload locations in host memory to NIC outbound buffer 246 using DMA transfers, as depicted by a block 608.

[0053] Meanwhile, TCP output processing continues until it is completed, as depicted by a block 610. Subsequent to TCP output processing completion, the logic proceeds to a block 612, wherein the context is updated with the processing

results and written back to the TCB cache 216. In a block 612, the scheduler 214 then updates completion queue 226 with completion descriptors and exception/event queue 228 with completion status data to signal end of transmit.

5 **[0054]** Figure 7 shows sets of general purpose instructions 700 and special purpose instructions 702 employed by one embodiment of the TOE. The general purpose instructions are roughly analogous to basic instructions used by general purpose processors, such as loading moving, simple mathematical operations, jumps, NOPs, etc.). In one embodiment, the general purpose instructions 700 operate on 32 bit operands.

10 **[0055]** The specialized instruction set was developed for efficient TCP processing. It includes special purpose instructions for accelerated context lookup, loading and write back. In one embodiment, these instructions enable context loads and stores from TCB cache 216 in eight slow cycles, as well as 512B wide context read and write between the core and thread cache 316 in a single core cycle. The
15 special purpose instructions include single cycle hashing, DMA transmit and receive instructions and timer commands. Hardware assist for conversion between host and network byte order is also available.

TCP-aware hardware multi-threading and scheduling logic

20 **[0056]** The multi-threaded architecture enables hiding of latency from memory accesses and other hardware functions and thus expedites inbound and outbound packet processing, minimizing the need for costly buffering and queuing. Hardware-assisted multi-threading enables storage of thread state in private (i.e., local host) memory. True hardware multi-threading takes this further by implementing the multiple thread mechanism entirely in hardware. In one embodiment, scheduler 214
25 is a TCP-aware scheduler that is configured to handle the tasks of thread suspension, scheduling, synchronizing and save/restore of thread state and the conditions that trigger them. TCP stack analysis shows that there are a finite

number of such conditions, which can be safely moved to hardware. The motivation is to free the programmer from the responsibility of maintaining and scheduling threads and to mitigate human error. This model is thus simpler than the more common conventional model of a programmer or compiler generated multi-threaded software code. In addition, the same code that runs on a single-threaded engine can run unmodified on processing engine 212 with greater efficiency. Under the hardware-based thread control architecture, the overhead penalty from switching between threads is kept minimal to achieve better throughput. In one embodiment, the architecture also provides instructions to support legacy manual multi-threaded programming.

[0057] Hardware multi-threading is best illustrated with an example. TCP packet processing requires several memory accesses as well as synchronization points with the DMA engine that can cause the execution core to stall while waiting for a response from such long-latency operations. Six such trigger conditions are identified (labeled A-F) in the pipeline diagrams in Figures 4a and 4b. If core TCP input or output processing completes prior to the parallel DMA operations, thread switch can occur to improve throughput. When the DMA operations end, the thread switches back to update the context with processed results and the updated context is written back to the TCB. Thread switches can happen during both transmit and receive processing. Unlike conventional software-based multi-threading, where thread switch, lock/un-lock and yield points are manually controlled, the TCP-aware scheduler controls the switching and synchronization between different threads in all the above cases.

[0058] In one embodiment, a single thread is associated with each network packet that is being processed, both incoming and outgoing. This differs from other conventional approaches that associate threads with each task to be performed, irrespective of the packet. The scheduler 214 spawns a thread when a packet

belonging to a new connection needs to be processed. A second packet for that same connection will not be assigned a thread until the first packet is completely processed and the updated context has been written back to TCB cache 216. This is under the control of scheduler 214. When the processing of a packet in the core is stalled, the thread state is saved in thread cache 316 and scheduler 214 will spawn a thread for a packet on a different connection. It may also wake up a thread for a previously suspended packet by restoring its state and allow it to run to completion. In one embodiment under this approach, scheduler 214 also spawns special maintenance threads for global tasks, such as gathering statistics on Ethernet traffic.

[0059] In one embodiment, scheduler 214 implements a priority mechanism to determine which packet to schedule next for core processing. Details of a block architecture for one embodiment of scheduler 214 that supports the priority mechanism is shown in Figure 8. The scheduler includes a completion events queue 800, a new packets queue 802, and a maintenance events queue 804. Each of these queues is processed by a control block 806. In one embodiment, the control block comprises a finite state machine. The control block 806 interfaces with processing engine 212 to provide core control.

[0060] In one embodiment, the priority mechanism is programmed into scheduler 214. In one embodiment, scheduler 214 arbitrates between events that wake up or spawn threads from the following categories:

1. New packets on fresh connections, or on existing connections with no active packets in the engine.
2. New packets on existing network connections with active packets in the engine.
3. Completion events for suspended threads.
4. Maintenance and other global events.

[0061] Efficient multi-threading is critical to the ability of the offload engine to scale up to multi-gigabit Ethernet rates. The design and validation of the TOE is simpler in this approach than conventional approaches to multi-threading. It also simplifies requirements on the compiler and the programming model.

5 **[0062]** As discussed above, embodiments of the invention may be implemented via an integrated circuit (i.e., semiconductor chip). In one embodiment, the TOE circuitry and DMA circuitry are implemented on a common platform chipset component, such as but not limited to an MCH. In other embodiments, the foregoing TOE functionality may be implemented via appropriate circuitry integrated into a NIC
10 or a processor.

[0063] In addition, embodiments of the present description may be implemented not only within a semiconductor chip but also within machine-readable media. For example, the designs described above may be stored upon and/or embedded within machine readable media associated with a design tool used for designing
15 semiconductor devices. Examples include a netlist formatted in the VHSIC Hardware Description Language (VHDL) language, Verilog language or SPICE language. Some netlist examples include: a behavioral level netlist, a register transfer level (RTL) netlist, a gate level netlist and a transistor level netlist. Machine-readable media also include media having layout information such as a GDS-II file.
20 Furthermore, netlist files or other machine-readable media for semiconductor chip design may be used in a simulation environment to perform the methods of the teachings described above.

[0064] The TOE described above has been architected for efficient TCP termination in the platform chipset. An analysis of the performance of such a system
25 has been modeled to predict its capability in terms of full duplex Ethernet bandwidth for particular packet sizes. Predicted bandwidth vs. packet size performance

corresponding to exemplary performance models for a single thread and multi-threaded architectures are shown in Figure 9.

5 **[0065]** Assuming the time for processing transmits is similarly distributed, the TOE multiplexes between receiving and transmitting packets. The bandwidth it can support is inversely proportional to the size of the packets, as shown in Figure 9. This analysis shows that the multi-threaded architecture is capable of wire-speed TCP termination at full duplex 10Gbps rate for packets larger than 289 bytes. A single-threaded design can achieve the same performance for packet sizes larger than 676 bytes, showing greater than 2X difference in performance.

10 **[0066]** The above description of illustrated embodiments of the invention, including what is described in the Abstract, is not intended to be exhaustive or to limit the invention to the precise forms disclosed. While specific embodiments of, and examples for, the invention are described herein for illustrative purposes, various equivalent modifications are possible within the scope of the invention, as
15 those skilled in the relevant art will recognize.

[0067] These modifications can be made to the invention in light of the above detailed description. The terms used in the following claims should not be construed to limit the invention to the specific embodiments disclosed in the specification and the claims. Rather, the scope of the invention is to be determined entirely by the
20 following claims, which are to be construed in accordance with established doctrines of claim interpretation.